UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

COMPARATIVA D'IMPLEMENTACIÓ HARDWARE VS SOFTWARE EN UN SISTEMA DE PROCESSAT D'IMATGE PER A DETECCIÓ DE CARES

Treball Final de Grau

Presentat a la Facultat de

l'Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

per

Joan Marimon Illana

En compliment parcial

dels requisits pel grau en

ENGINYERIA DE SISTEMES ELECTRÒNICS

Tutor: Francesc Moll Echeto

Barcelona, Juny 2016

Abstract

In this project two implementations of a face detection system, one of them based on hardware and the other on software, have been developed. The main goal is to compare both implementations and determine the advantages or disadvantages of using a dedicated hardware.

The detection algorithm that has been used is a Haar-like features classifier based on the method proposed by Viola and Jones to accelerate object detection processing. The original code of the algorithm and the classifier that has been used were obtained from the OpenCV library. Vivado HLS has been used in order to synthetize the C/C++ code to generate an RTL description and implement the hardware version of the system.

The project has been developed in different stages. At first the original code was analysed in order to obtain a simplified version which could be used to develop both the hardware and software implementations. Finally the results of those implementations were compared, analysing also the efficiency of Vivado HLS as a development tool.

Resum

Aquest treball consisteix en la realització de dues implementacions d'un mateix sistema de detecció de cares, una basada en hardware i l'altra en software, amb l'objectiu de comparar-les i avaluar els beneficis o inconvenients d'utilitzar un hardware dedicat.

L'algoritme de detecció que s'ha utilitzat és un classificador de característiques pseudo-Haar basat en el mètode proposat per Viola i Jones per accelerar el procés de detecció d'objectes. Tant el codi original de l'algoritme com les dades del classificador utilitzades s'han obtingut de la llibreria OpenCV. També s'ha utilitzat Vivado HLS, que permet sintetitzar codi en C/C++ per generar directament una descripció RTL, per realitzar la implementació en hardware.

El projecte s'ha dut a terme en diferents etapes. La primera part consisteix en una anàlisi i simplificació del codi per a continuació realitzar les diferents implementacions. Finalment s'ha realitzat la comparativa entre la versió en hardware i software conjuntament amb una anàlisi de l'eficiència de Vivado HLS com a eina de desenvolupament.

Resumen

Este trabajo consiste en la realización de dos implementaciones distintas para un mismo sistema de detección de caras, una basada en hardware i la otra en software, con el objetivo de compararlas i determinar los beneficios o inconvenientes de usar un hardware dedicado.

El algoritmo de detección que se ha usado consiste en un clasificador de características pseudo-Haar basado en el método propuesto por Viola y Jones para acelerar el proceso de detección de objetos. Tanto el código original del algoritmo como los datos del clasificador utilizados se han obtenido de la librería OpenCV. También se ha utilizado Vivado HLS, que permite sintetizar código en C/C++ para generar directamente una descripción RTL, para realizar la implementación en hardware.

El proyecto se ha llevado a cabo en distintas etapas. En la primera parte se ha analizado un análisis i simplificación del código para, a continuación, realizar las distintas implementaciones. Finalmente se han comparado los resultados obtenidos en la versión hardware y software al mismo tiempo que se ha analizado la eficiencia de Vivado HLS como herramienta de desarrollo.

Dedicat a la Leona, per mossegar-me quan treballo.

Agraïments

En primer lloc m'agradaria agrair al meu tutor Francesc Moll, des de la seva ajuda en l'elecció del tema del treball fins al seu suport i interès durant el desenvolupament del projecte. En segon lloc agrair a la meva família per fer possible que continués estudiant i donar-me suport al llarg dels anys.

Historial de revisions i registre d'aprovacions

Revisió	Data	Objectiu
0	25/05/2016	Creació del document
1	14/06/2016	Revisió del document
2	21/06/2016	Revisió final

Llista de participants

Nom	e-mail
Joan Marimon	joan.marimon@alu-etsetb.upc.edu
Francesc Moll	francesc.moll@upc.edu

Escrit per:		Revisat i aprovat per:	
Data	21/06/2016	Data	22/06/2016
Nom	Joan Marimon	Nom	Francesc Moll
Posició	Autor del projecte	Posició	Supervisor del projecte

Taula de continguts

Llista de Figures

Llista de Taules:

1. Introducció

El treball consisteix en la realització de dues implementacions d'un sistema de detecció de cares, una basada en hardware i l'altra en software, amb l'objectiu de comparar-les i avaluar els beneficis o inconvenients d'utilitzar un hardware dedicat.

Per dur a terme el projecte s'utilitzarà la placa de desenvolupament ZedBoard que està basada en un dispositiu Zynq de Xilinx el qual conté un processador i una FPGA integrats en el mateix xip. A més es partirà d'un algoritme de detecció de cares ja existent basat en el mètode proposat per Viola i Jones i implementat en la llibreria OpenCV. A partir d'aquí s'utilitzarà Vivado HLS per sintetitzar el codi i obtenir una descripció en hardware. D'aquesta manera, a part de realitzar una comparativa entre les implementacions en hardware i software, també es comprovarà l'eficiència de Vivado HLS en quan a millora en el temps de disseny i desenvolupament d'aplicacions en hardware.

L'objectiu final és implementar un sistema format per una càmera, un sistema de processar (la placa de desenvolupament ZedBoard) i una pantalla on visualitzar els resultats que ha de ser capaç de captar i processar les dades suficientment ràpid com per implementar una detecció de cares en temps real.

1.1. Pla de treball

<table>
<tr><td colspan="2">Projecte: Comparativa d'implementació Hardware vs Software en un sistema de processat d'imatge per a detecció de cares.</td><td>WP ref: (WP1)</td></tr>
<tr><td colspan="2">Major constituent: Selecció de components.</td><td>Full 1 de 6</td></tr>
<tr><td colspan="2">Descripció:
Determinar els components a utilitzar per a la realització del projecte.</td><td>Data d'inici: 15/02/2016
Data de finalització: 17/02/2016</td></tr>
<tr><td>Tasca interna T1: Estudi dels requeriments del sistema.
Tasca interna T2: Seleccionar la càmera.
Tasca interna T3: Seleccionar la interfície de pantalla (VGA o HDMI).</td><td>Entregables:</td><td>Dates:</td></tr>
</table>

Projecte: Comparativa d'implementació Hardware vs Software en un sistema de processat d'imatge per a detecció de cares.	WP ref: (WP2)	
Major constituent: Algoritme.	Full 2 de 6	
Descripció: Determinar l'algoritme de detecció de cares que s'utilitzarà i obtenir una primera implementació en software.	Data d'inici: 18/02/2016 Data de finalització: 03/03/2016	
Tasca interna T1: Determinar l'algoritme a utilitzar. Tasca interna T2: Descripció detallada de l'algoritme. Tasca interna T3: Implementar l'algoritme en llenguatge d'alt nivell (C/C++ amb OpenCV).	Entregables: Implemenació de l'algoritme	Dates: 03/03/2016

Projecte: Comparativa d'implementació Hardware vs Software en un sistema de processat d'imatge per a detecció de cares.	WP ref: (WP3)	
Major constituent: Implementació hardware.	Full 3 de 6	
Descripció: Obtenir un sistema de detecció de cares implementat en hardware.	Data d'inici: 4/03/2016 Data de finalització: 28/04/2016	
Tasca interna T1: Implementar interfície entre la càmera i el sistema de processat. Tasca interna T2: Implementar interfície entre el sistema de processat i pantalla. Tasca interna T3: Gestionar el tractament de dades i memòria del sistema de processat. Tasca interna T4: Adaptació de l'algoritme per a la implementació en hardware (HLS). Tasca interna T5: Visualització dels resultats.	Entregables: Projecte Vivado Projecte vivado HLS	Dates: 03/05/2016 03/05/2016

Projecte: Comparativa d'implementació Hardware vs Software en un sistema de processat d'imatge per a detecció de cares.	WP ref: (WP4)	
Major constituent: Implementació software.	Full 4 de 6	
Descripció: Obtenir un sistema de detecció de cares implementat en software.	Data d'inici: 04/05/2016 Data de finalització: 20/05/2016	
Tasca interna T1: Implementar interfície entre la càmera i el sistema de processat Tasca interna T2: Implementar interfície entre el sistema de processat i pantalla Tasca interna T3: Gestionar el tractament de dades i memòria el sistema de processat Tasca interna T4: Adaptació de l'algoritme per a la implementació en software Tasca interna T5: Visualització dels resultats	Entregables: Projecte Vivado Projecte Vivado SDK	Dates: 20/05/2016 20/05/2016

Projecte: Comparativa d'implementació Hardware vs Software en un sistema de processat d'imatge per a detecció de cares	WP ref: (WP5)	
Major constituent: Optimització de la implementació en hardware	Full 5 of 6	
Descripció: Modificar el codi en C/C++ i les directrius de síntesis en Vivado HLS per tal d'optimitzar la implementació en hardware.	Data d'inici: 23/05/2016 Data de finalització: 10/06/2016	
Tasca interna T1: Modificar el codi per afavorir la paral·lelització del sistema Tasca interna T2: Modificar les directrius de síntesis de Vivado HLS per tal d'optimitzar la implementació en hardware.	Entregables: Projecte Vivado HLS	Dates: 10/06/2016

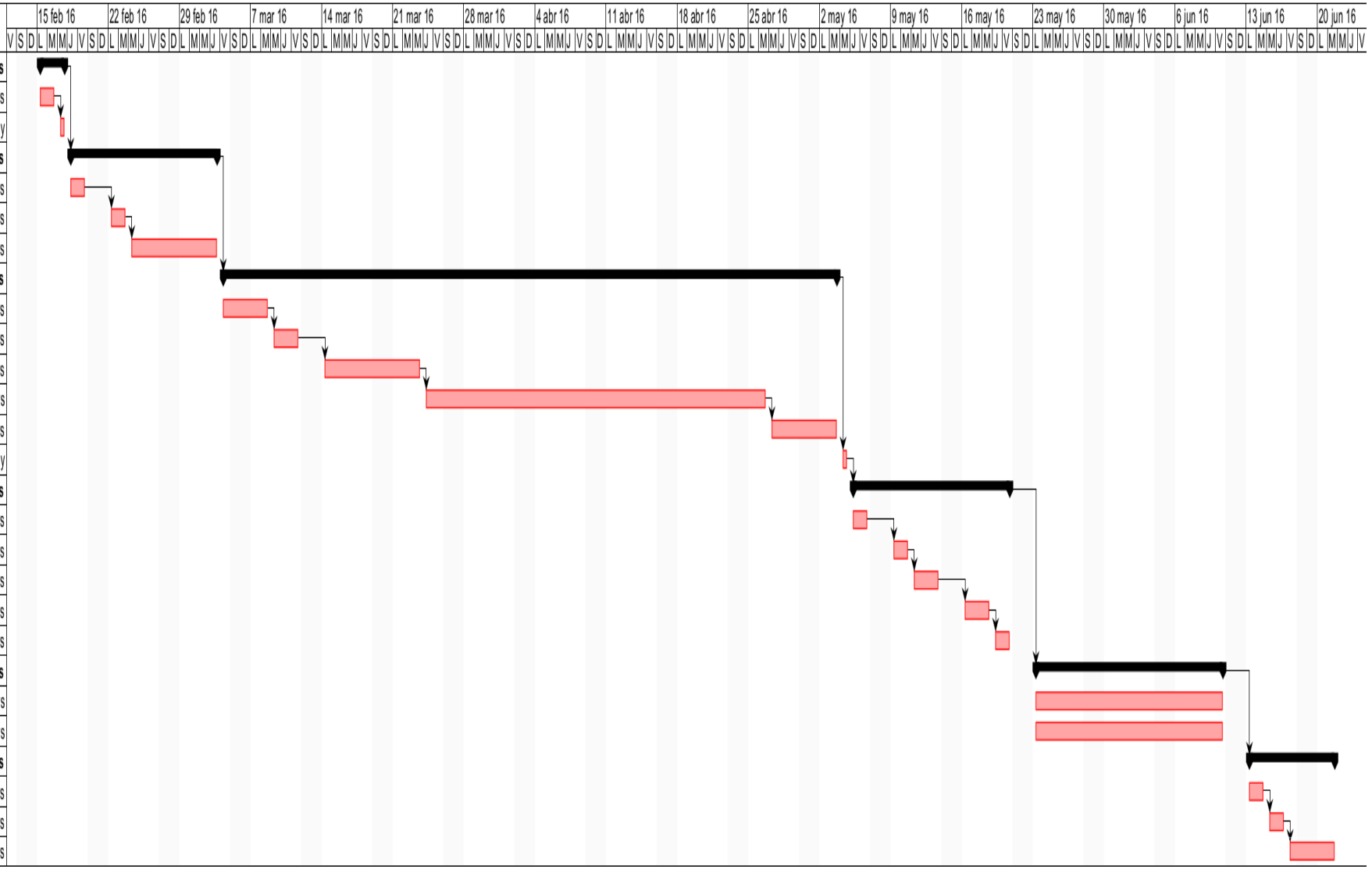
<table>
<tr><td colspan="2">Projecte: Comparativa d'implementació Hardware vs Software en un sistema de processat d'imatge per a detecció de cares</td><td>WP ref: (WP6)</td></tr>
<tr><td colspan="2">Major constituent: Comparativa</td><td>Full 6 de 6</td></tr>
<tr><td colspan="2" rowspan="2">Descripció:
Obtenir conclusions a partir de la avaluació i comparació dels resultats obtinguts</td><td>Data d’inici: 13/06/2016
Data de finalització: 21/06/2016</td></tr>
<tr><td>Start event: 21
End event: 24</td></tr>
<tr><td>Tasca interna T1: Comparar el funcionament del sistema en les dues implementacions
Tasca interna T2: Comparar rendiment i velocitat del processat d'imatge en les dues implementacions
Tasca interna T3: Comparar el consum de recursos i eficiència del sistema en les dues implementacions</td><td>Entregables:
Conclusions de la comparativa</td><td>Dates:
21/03/2016</td></tr>
</table>

Fites

WP#	Tasca#	Títol	Entregables	Setmana
WP1	T3	Selecció de components	Determinar components	1
WP2	T3	Implementació de l’algoritme	Test de l’algoritme	2
WP2	T3	Implementació de l’algoritme	Programa en C/C++	3
WP3	T4	Implementació hardware	Test de funcionament hw	10
WP3	T5	Implementació hardware	Projectes Vivado	12
WP4	T4	Implementació software	Test de funcionament sw	14
WP4	T5	Implementació software	Projectes Vivado	14
WP5	T2	Optimització	Projectes Vivado	17
WP6	T3	Comparativa	Elaborar conclusions	19

1.2. Diagrama de Gantt

	Nombre	Duracion
1	**Selecció de components**	**3 days**
2	Estudi de requeriments del sistema	2 days
3	Seleccionar components	1 day
4	**Algoritme**	**11 days**
5	Determinar l'algoritme	2 days
6	Descripció de l'algoritme	2 days
7	Implementar algoritme en alt nivell	7 days
8	**Implemenació hardware**	**43 days**
9	Interfície càmera	3 days
10	Interfície pantalla	3 days
11	Gestió memòria	8 days
12	Adaptar algoritme per a hardware	24 days
13	Implementar visualització de resultats	5 days
14	Revisió crítica	1 day
15	**Implementació software**	**12 days**
16	Interfície càmera	2 days
17	Interfície pantalla	2 days
18	Gestió memòria	3 days
19	Adaptar algoritme	3 days
20	Visualització de resultats	2 days
21	**Optimització**	**15 days**
22	Modificació del codi	15 days
23	Ajustar paràmetres de síntesis	15 days
24	**Comparativa**	**7 days**
25	Comparar funcionament	2 days
26	Comparar rendiment	2 days
27	Comparar recursos	3 days



1.3. Modificacions respecte el pla de treball original

Durant la implementació en hardware han sorgit diversos problemes que han causat que no es pogués complir el termini establert en el pla de treball original:

- Errors en la visualització dels resultats per pantalla degut a una mala gestió de les dades i la memòria.
- Durant l'adaptació de l'algoritme per a la implementació en hardware (HLS) s'ha hagut de modificar i reorganitzar el codi més del que s'havia previst en un principi.
- Els resultats obtinguts en una primera implementació en hardware no eren tan bons com s'esperaven i s'ha hagut de destinar més temps a l'optimització.

Degut als problemes per obtenir una primera implementació en hardware es va allargar el termini destinat a adaptar l'algoritme una setmana. A causa d'això va reorganitzar el pla de treball de cara al final del projecte. En un principi s'havia destinat prop d'un mes per a la implementació en software però en el nou pla de treball es redueix a dos setmanes per tal de tenir temps per optimitzar la implementació en hardware.

2. Tecnologies aplicades

En els últims anys, en part motivat per l'auge de la realitat augmentada i els sistemes autòmats intel·ligents, s'han incrementat considerablement les aplicacions de visió per ordinador on un processador ha de ser capaç de detectar i reconèixer objectes en una imatge. Concretament una de les aplicacions més utilitzades en diversos camps és la de detecció de cares. La detecció de cares s'utilitza generalment com a primera etapa per a un posterior processat més complex com pot ser el de reconeixement de cares, detecció de parpelleig i cansament o per determinar l'edat i el sexe d'una persona.

A continuació es farà una introducció a un del algoritmes de detecció d'objectes més utilitzats, que en el nostre cas s'aplicarà a detectar cares, així com al dispositiu i al software que s'utilitzarà durant el desenvolupament del projecte.

2.1. Algoritme de detecció de cares

Els algoritmes de detecció de cares es basen en el càlcul de certes característiques d'una imatge per determinar si es tracta d'una cara o no. Un dels mètodes més utilitzats és el proposat per Viola i Jones [7] que aporta tres idees que permeten accelerar el procés de detecció d'objectes considerablement.

Figura 1: Característiques pseudo-Haar

Com s'ha comentat, les imatges es classifiquen segons el valor de característiques simples, anomenades pseudo-Haar[1]. En l'algoritme de Viola i Jones s'utilitzen tres tipus de característiques formades per dos, tres o quatre rectangles adjacents entre si. Prenent com a referència la Figura 1, el valor d'una característica correspon a la suma dels píxels de les regions negres menys la suma dels píxels de la regions blanques.

Per calcular ràpidament aquestes característiques, Viola i Jones, van proposar la imatge integral on cada posició x, y correspon a la suma dels valors dels píxels a sobre i a l'esquerra de x', y' (ambdós inclosos) en la imatge original:

$$ImatgeIntegral(x,y) = \sum_{\substack{x' \le x \\ y' \le y}} ImatgeOriginal(x',y')$$

D'aquesta manera es pot realitzar la suma dels píxels de qualsevol regió rectangular amb tan sols quatre accessos a memòria. Com es pot veure en la Figura 2, la suma dels píxels de la regió A correspon al valor de la imatge integral en la posició 1, la posició 2 és la suma de $A+B$, la posició 3 correspon a la suma de $A+C$ i la posició 4 és $A+B+C+D$. La suma dels píxels de la regió D es pot calcular com $II(4)+II(1)-II(2)-II(3)$, on II correspon a la imatge integral.

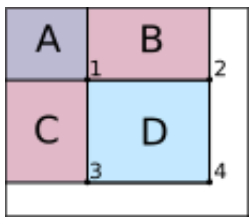


Figura 2: Càlcul de la suma dels píxels d'una regió amb la imatge integral

[1] *Haar-like features*

Viola i Jones van determinar que no cal calcular totes les característiques sinó que un grup reduït és suficient per detectar l'objecte. Per trobar aquestes característiques es parteix d'un conjunt d'entrenament format per un gran nombre d'imatges positives i negatives al que s'aplica un algoritme d'aprenentatge automàtic[2].

En aquest cas van utilitzar l'algoritme AdaBoost per trobar els *classificadors febles*, formats per una sola característica, que millor separen les mostres positives de les negatives. Per cada un dels classificadors febles es calcula el valor llindar òptim que minimitza el nombre de mostres mal classificades. D'aquesta manera un classificador feble $h_j(x)$, on x correspon a una finestra de 20x20 píxels, està format per una característica f_j, un valor llindar θ_j i una paritat p_j que indica la direcció de la inequació:

$$h_j(x) = \begin{cases} 1, & p_j \cdot f_j(x) < p_j \cdot \theta_j \\ 0, & altres \end{cases}$$

Una sola característica no pot classificar les imatges de forma correcta. Per tal d'implementar una detecció ràpida i precisa Viola i Jones van proposar un classificador en cascada (Figura 3) format per diferents etapes on cada etapa descarta gran part de les mostres negatives al mateix temps que detecta la majoria de mostres positives.

Cada etapa, també anomenada classificador fort, està formada per diversos classificadors febles. Un altre cop s'utilitza l'algoritme AdaBoost per seleccionar el conjunt de classificadors febles de cada etapa i determinar el valor llindar que minimitza els falsos positius. Es configuren les primeres etapes per tal que tinguin poques característiques, de manera que es pugin calcular ràpidament, i al mateix temps descartin la major part de mostres negatives. Les etapes posteriors són cada cop més complexes de manera que com més etapes superi una finestra més probable és que correspongui a una cara. Així si una finestra supera totes les etapes l'algoritme determina que allà hi ha una cara.

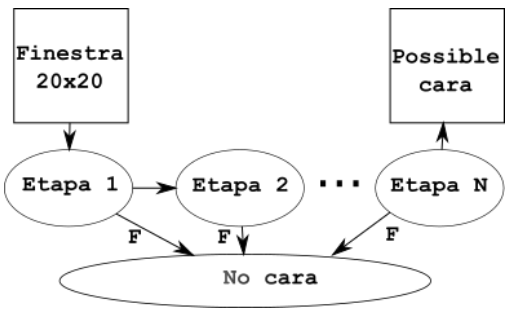


Figura 3: Esquema d'un classificador en cascada

Per minimitzar els efectes de la il·luminació les imatges de cares utilitzades en l'entrenament eren de variància normalitzada. Per tant també s'ha de normalitzar la variància en la detecció. Tenint en compte la fórmula de la variància:

$$\sigma^2 = m^2 - \frac{1}{N}\sum x^2$$

on σ correspon a la desviació típica, m al valor mig i x al valor del píxels de la finestra, es pot calcular el valor de la variància a partir de dos imatges integrals. El valor mig es s'obté de la imatge integral original i la suma dels píxels al quadrat s'obté d'una nova imatge integral calculada a partir de la imatge original al quadrat.

[2] *Machine learning*

2.1.1. OpenCV

OpenCV [5] és una llibreria de codi obert que implementa diferents algoritmes de visió per ordinador i aprenentatge automàtic. Està implementada originalment en C++ però ofereix interfícies tant per Python, Java o C.

L'algoritme de detecció de cares és bastant complex, sobretot pel fet que es necessita un bon classificador per tal que la detecció sigui precisa. Per obtenir un bon classificador es necessiten moltes mostres positives, de l'ordre de milers, a més de configurar correctament l'algoritme d'entrenament. Per aquest motiu s'ha optat per utilitzar l'algoritme i un dels classificador ja entrenats de la llibreria OpenCV.

L'algoritme de l'OpenCV que s'utilitzarà és un classificador de característiques pseudo-Haar basat en el mètode proposat per Viola i Jones descrit anteriorment. Concretament el classificador fa servir característiques de dos i tres rectangles, 2135 classificadors febles organitzats en 22 etapes i s'ha entrenat amb una finestra de 20x20 píxels.

2.2. ZedBoard™

La ZedBoard és una placa d'avaluació i desenvolupament basada en un Xilinx Zynq™-7000 All Programmable SoC (AP SoC). En el mateix xip es combinen un processador amb lògica programable, concretament el xip és el XC7Z020-1CLG484C i conté un processador Dual ARM® Cortex™-A9 MPCore™ amb 85.000 cèl·lules programables de Serie-7 de Xilinx. A més a més la ZedBoard està equipada amb tot un seguit de perifèrics i capacitat d'expansió que la converteixen en una eina versàtil amb la que es poden dur a terme una gran varietat de projectes.

Pel que fa als perifèrics, els més rellevants per al projecte són els connectors PMOD per connectar una càmera, la interfície de vídeo VGA per visualitzar els resultats i la memòria DDR3 de 512 MB. Addicionalment també es podrà fer servir la interfície USB-UART per establir una comunicació directa entre l'ordinador i la placa.

Donades les seves característiques la ZedBoard permetrà implementar l'algoritme de detecció de cares tant en software com en hardware a més de poder realitzar una comparativa entre dispositius tecnològicament equivalents entre si.

Dins la família de dispositius Zynq el xip XC7Z020-1CLG484C es pot definir com a un dispositiu de gamma mitja/baixa. Donat que en aquest projecte es pretén diferenciar entre una implementació en software i una altra en hardware en els següents apartats s'analitzaran, per separat, les característiques del xip en quant a processador i lògica programable tot comparant-les amb la tecnologia actual.

2.2.1. Processador (PS)

L'ARM Cortex-A9 [13] és un processador de 32 bits d'ús general, baix consum i cost moderat. És un dels processadors ARM més utilitzats i està present una gran varietat de dispositius d'aplicacions diverses. Fins fa pocs anys era un dels processadors més utilitzats en els telèfons intel·ligents. Dins dels processadors ARM la sèrie Cortex-A és la que té més capacitat de processament.

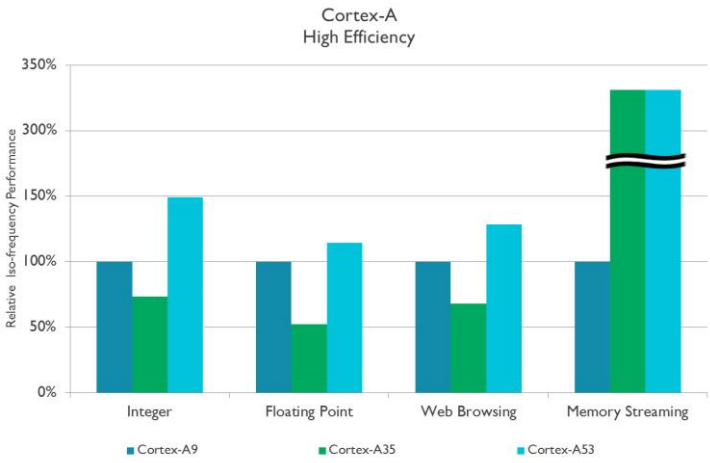


Figura 4: Comparativa entre processadors de la família Cortex-A [12]

Els processadors Cortex-A [12] es divideixen en tres grups: alt rendiment, alta eficiència i ultra-alta eficiència. En aquesta classificació el Cortex-A9 correspon al grup d'alta eficiència, definint-se com un processador de classe mitja dins la sèrie Cortex-A. La Figura 4 correspon a una comparativa del rendiment entre el xip Cortex-A9, un xip de ultra-alta eficiència (Cortex-A35) i un xip d'alt rendiment (Cortex-A53) operant a la mateixa freqüència.

2.2.2. Lògica programable (PL)

La lògica programable del el xip XC7Z020-1CLG484C és equivalent a la d'una FPGA de gamma mitja de la sèrie Artix-7 [2].

Concretament disposa de 85.000 cèl·lules lògiques, 53.200 LUT, 106.400 Flip-Flops, 560 KB de memòria RAM (organitzats en 140 blocs de 36 KB o 280 de 18 KB) i 220 DSP.

En quant a recursos disponibles la lògica del xip de la ZedBoard és aproximadament 2,5 vegades inferior a la lògica del xip amb més recursos de la família Artix-7. A més a més, dins la sèrie 7 de FPGAs de Xilinx, la família Artix-7 és la de gamma més baixa i és la que ofereix els dispositius de cost i potència més reduïts. Les altres famílies són la Kintex-7 i la Virtex-7, cada una amb el doble de recursos que l'anterior [3]. Així doncs, en quant a lògica la ZedBoard té uns recursos limitats tenint en compte les alternatives disponibles.

2.2.3. Interfície PS-PL

Un dels aspectes més interessants dels xips Zynq és el fet que combinen un processador amb lògica programable i, a més a més, diferents ports que permeten establir una comunicació entre les dues parts.

Alguns dels elements de la interfície PS-LS són ([2] i [8]):

- Interfície AMBA AXI per la comunicació de dades:
  - Quatre interfícies AXI de 32 bits que permeten comunicació directa entre PS i PL, definides com a ports de propòsit general (AXI GP).
  - Quatre interfícies AXI de 64/32 bits amb accés directe a la memòria DDR, definides com a ports d'alt rendiment (AXI HP).

- Rellotges i resets:
    - Quatre rellotges i resets generats en el processador amb sortida directa a la lògica programable.

2.3. Vivado Design Suite

Vivado Design Suite [14] és el software de Xilinx per al desenvolupament amb FPGA. Està dissenyat per augmentar la productivitat en el disseny, integració i implementació de sistemes basats en els xips de la sèrie 7 de Xilinx, Zynq-7000 All Programmable (AP) SoC i dispositius UltraScale. Conté les eines necessàries per dissenyar, simular, sintetitzar, implementar analitzar i programar el hardware.

La nova versió, Vivado Design Suite HLx, proporciona una metodologia pensada per augmentar la productivitat.. Aquesta versió inclou Xilinx SDK, Vivado Hight-Level Synthesis (HLS) a més de llibreries C/C++ i Vivado IP Integrator.

Vivado HLS permet sintetitzar funcions en C/C++ directament a descripcions RTL en VHDL o Verilog. A més a més, abans de sintetitzar, Vivado HLS ofereix la possibilitat de realitzar una simulació funcional del sistema directament en C/C++, que és significativament més ràpida que una simulació d'una descripció RTL. També permet optimitzar el disseny segons les necessitats del sistema en quan a velocitat o recursos i, un cop sintetitzat, ofereix una estimació dels recursos i latència del sistema.

Xilinx SDK proporciona un entorn de desenvolupament per crear aplicacions de software per als processadors encastats de Xilinx, com per exemple els processadors ARM dels dispositius Zynq. Els projectes de l'SDK es creen a partir d'un disseny en hardware prèviament creat amb Vivado Design Suite. A més d'un editor de codi l'SDK també té eines per fer debug i analitzar l'eficiència del codi.

Xilinx ofereix una versió gratuïta del software, Vivado Design Suite HL WebPACK, que està limitada als dispositius Artix®-7 (7A35T - 7A200T), Kintex®-7 (7K70T, 7K160T) i Zynq®-7000 All Programmable SoC Devices (XC7Z7010 - XC7Z7030). En quest projecte s'utilitza la versió 2015.4 de Vivado Desing Suite HL WebPACK sobre un ordinador amb processador Intel Core i7 de 8 nuclis a 2.5 GHz amb 8 GB de memòria RAM i el sistema operatiu Ubuntu 14.04 LTS.

2.3.1. Vivado HLS

Com s'ha comentat Vivado HLS permet sintetitzar codi en C/C++ per obtenir descripcions RTL amb unes mínimes modificacions. En aquest apartat es farà una introducció a les funcionalitats de Vivado HLS. Si es vol una informació més completa es pot consultar la guia d'usuari [10].

Per defecte Vivado HLS sintetitza els paràmetres de les funcions com a ports en una descripció RTL i les funcions es sintetitzen com a blocs dins la jerarquia del RTL (per exemple si una funció conté sub-funcions, questes es sintetitzaran com mòduls o entitats en la descripció RTL final, seguint la mateixa jerarquia que el codi en C/C++ original).

A més proporciona informes sobre el rendiment del sistema, principalment sobre els recursos utilitzats, la latència (definida com els cicles que tarda una funció per calcular

totes les dades) i l'interval d'iniciació[3] (definit com el número de cicles que han de passar perquè la funció accepti noves dades d'entrada).

Vivado HLS proporciona a més tot un seguit de llibreries les quals permeten crear codi en C/C++ que correspon de forma directa amb estructures de hardware bàsiques.

Una de les llibreries proporciona variables de mida arbitrària, és a dir permet definir el número de bits de les variables. En el cas de C++ la llibreria que s'ha d'incloure és "ap_int.h" i proporciona les següents classes:

- ap_[u]int<N>: Defineix una variable entera de N bits, que pot ser amb signe o sense signe.
- ap_[u]fixed<N,E>: Defineix una variable decimal de punt fix de N bits amb E bits per a la part entera. També pot ser amb signe o sense.
- half: Defineix una variable decimal de punt flotant de 16 bits.

Vivado HLS també disposa d'una llibreria matemàtica, "hls_math.h", que proporciona suport per a la síntesis de les funcions matemàtiques de les llibreries estàndard de C/C++ ("math.h" o "cmath.h"). Suporta una gran varietat de funcions des d'operacions bàsiques com el valor absolut o funcions trigonomètriques a operacions més complexes com arrels quadrades, exponents, logaritmes, operacions amb punt flotant, etc. Tot i així cal mencionar que no suporta totes les funcions incloses a les llibreries matemàtiques estàndard de C.

Finalment Vivado HLS proporciona directives que permeten configurar com es realitzarà la síntesis. Cal tenir en compte que les directives són tan sols indicacions i Vivado HLS pot ignorar-les si no fos possible aplicar-les. Algunes d'aquestes directives són:

- #pragma HLS pipeline: Aplica pipeline a un bucle de manera que es pot començar a executar la següent iteració mentre encara s'executa la primera.
- #pragma HLS inline: Per defecte quan es crea una funció en la descripció RTL es crea un mòdul. Aquesta directiva fa que no es creï cap mòdul per aquesta funció sinó que es descrigui la lògica directament.
- #pragma HLS unroll: Desfà un bucle, és a dir implementa totes les iteracions del bucle en paral·lel.
- #pragma HLS loop_tripcount: Vivado HLS realitza una estimació de la latència del sistema però no ho pot fer en els casos en que els límits d'iteració no estan definits en el moment de la síntesis. Amb aquesta directiva es poden definir els límits d'iteració per tal que Vivado HLS pugui realitzar l'estimació.

Vivado HLS no pot sintetitzar qualsevol codi, per a que el codi sigui sintetitzable s'ha de complir:

- El codi no ha de realitzar crides del sistema al sistema operatiu.
- Les variables han de ser de mida definida, no es poden utilitzar variables de memòria dinàmica.

[3] Initiation interval

3. Metodologia i desenvolupament del projecte:

El projecte s'ha dut a terme en diferents etapes. La primera part consisteix en un estudi del codi proporcionat per OpenCV amb l'objectiu d'entendre com s'implementa l'algoritme i obtenir una versió simplificada amb la qual treballar en les etapes posteriors. La segona part consisteix en sintetitzar el codi amb Vivado HLS per tal de realitzar una implementació en hardware de l'algoritme. Finalment en l'última part es programa l'algoritme en software.

3.1. Implementació de l'algoritme

Per tal d'obtenir l'algoritme de detecció de cares s'ha utilitzat la llibreria OpenCV. L'objectiu en aquest punt és entendre l'algoritme i determinar quines son les funcions necessàries per poder realitzar posteriorment la implementació en hardware.

La llibreria conté les funcions necessàries per implementar l'algoritme així com classificadors prèviament entrenats guardats en format xml.

En aquest cas s'ha utilitzat el classificador 'haarcascade_frontalface_alt.xml', disponible a "*OPENCVDIR/sources/data/haarcascades/*" on *OPENCVDIR* és el directori arrel de l'OpenCV. Com a imatge de prova s'han utilitzat versions modificades de la imatge 'lena.jpg', disponible a "*OPENCVDIR/sources/samples/data/".*

A mode de prova s'han realitzat algunes implementacions de l'algoritme en un ordinador sobre el sistema operatiu Ubuntu 14.04 LTS i s'ha utilitzat Netbeans 8.1 com a entorn de programació (IDE).

OpenCV és fàcil d'utilitzar i per tal d'implementar l'algoritme tan sols s'han de cridar dues funcions de la llibreria. S'ha creat un programa en C/C++ que, utilitzant la llibreria OpenCV, carrega una imatge sobre la qual s'aplicarà l'algoritme, carrega les dades del classificador des de l'arxiu xml, aplica l'algoritme i finalment mostra el resultat.

Al ser de codi obert es pot analitzar directament el codi de la llibreria OpenCV. La llibreria és bastant extensa, s'ha anat modificant i ampliant al llarg dels anys i, a causa d'això, no és fàcil trobar els fitxers on s'implementa l'algoritme de detecció de cares. Per tal de determinar el procés que fa el programa per implementar l'algoritme es va optar per fer debug i executar el codi línia a línia per entrar directament dins les funcions de la llibreria i veure realment quin codi s'executa. OpenCV utilitza per defecte diferents fils per a accelerar l'execució del programa però en aquest cas es va indicar que no s'utilitzessin fils i s'executés el programa en un sol procés per tal de facilitar l'anàlisi.

Primer carrega una imatge i la converteix a escala de grisos. A partir d'aquí genera la imatge integral per a cada escala i la guarda en un buffer de memòria (sBuf). A part de la imatge integral, en el mateix buffer també es guarda la suma dels píxels al quadrat per tal de calcular la variància. A continuació una finestra recorre les diferents imatges integrals, per detectar cares en diferents escales, i en cada posició aplica l'algoritme de detecció que calcula els valors dels diferents classificador febles corresponents a cada etapa. Finalment, cada cop que es troba una cara es guarda en un vector de memòria dinàmica.

Quan s'acaba el procés es poden agrupar els rectangles de tal manera que tan sols quedi un rectangle per cada cara detectada i dibuixar aquest rectangle sobre la imatge

original per indicar quines són les cares detectades. En la Figura 5 es pot veure un esquema simplificat d'aquest procés.

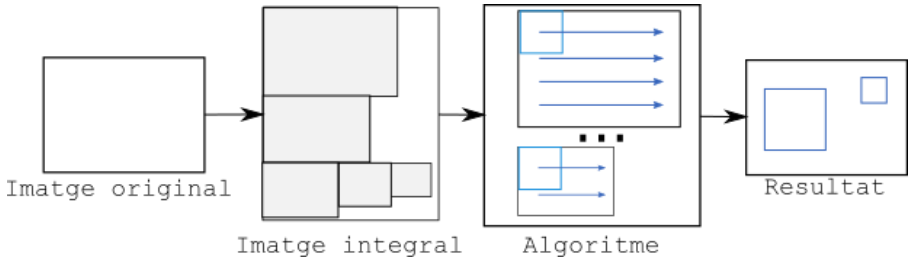


Figura 5: Esquema de l'algoritme de detecció de cares implementat per OpenCV

Un cop analitzat el codi es van extreure les funcions i classes necessàries per tal de crear un codi simplificat amb el qual es pogués treballar més fàcilment.

L'algoritme final utilitza un classificador de característiques pseudo-Haar format per un total de 2135 classificadors febles organitzats en 22 etapes. A més s'ha utilitzat un factor d'escala de 1,1 que, en el cas de la resolució utilitzada de 160x120, dóna lloc a 19 escales diferents.

3.2. Implementació en hardware

3.2.1. Síntesi

Vivado HLS permet sintetitzar el codi en C/C++ per generar una descripció de hardware amb la qual configurar la lògica programable. No es pot sintetitzar qualsevol codi ja que, com s'ha comentat, per tal que un codi sigui sintetitzable s'han de seguir certes regles. En aquest cas les regles que ens afecten són:

- S'han d'eliminar les funcions que depenen del sistema operatiu, per exemple les que accedeixen a fitxers o la implementació de fils.
- Totes les variables han de tenir la mida definida al moment de la síntesis. És a dir, Vivado HLS no permet sintetitzar vectors de memòria dinàmica.

Així dons, el primer pas per sintetitzar el codi és modificar totes les parts afectades per aquestes limitacions per tal que Vivado HLS sigui capaç de sintetitzar-lo.

Per eliminar la necessitat d'accedir a fitxers s'han extret les dades del fitxer 'haarcascade_frontalface_alt.xml' que descriuen les diferents etapes, el valor dels classificadors febles i els diferents rectangles que formen les característiques. Per fer-ho s'han creat, en el mateix codi C/C++, vàries funcions que obtenen les dades de les diferents característiques, etapes i escales i les exporten a fitxers de capçalera (.hpp) on es guarden les dades en forma de vectors. Aquests fitxers de capçalera s'inclouran després en el projecte de Vivado HLS.

També s'ha de passar d'utilitzar la llibreria matemàtica del sistema 'cmath' a la llibreria 'hls_math.h' proporcionada per Vivado HLS per implementar l'arrel quadrada en el càlcul de la variància de la finestra.

Per altra banda els vectors de memòria dinàmica es substitueixen per vectors de mida fixa. Per la majoria de vectors no hi ha problema ja que en sabem la mida. El problema

resideix en el vector que guarda les cares detectades ja que no sabem, a priori, les cares que es detectaran. En aquest cas es van pensar dues solucions:

- Limitar el nombre de cares detectades. En aquest cas la solució és simplement utilitzar un vector de mida fixa suficientment gran com per guardar les dades de les cares que s'esperen detectar (tenint en compte que l'algoritme pot detectar una mateixa cara vàries vegades).
- Enquadrar la cara directament. En aquest projecte no és necessari determinar la posició de la cara sinó que simplement es vol detectar, per això en aquesta solució simplement es dibuixa un quadrat al detectar una cara. Al no guardar cap dada no es necessita cap vector i, per tant, no hi ha límit de cares que es poden detectar.

Variable	Número d'elements	Tipus de variable	Número de BRAM	Tipus de variable	Número de BRAM
sBuf	292608	Int	521	ap_uint<31>	504
RECTS_ARRAY	25620	Int	46	ap_uint<5>	8
WEIGHTS_ARRAY	6405	Int	12	ap_int<3>	2
NODES_ARRAY	2135	Int	4	ap_uint<12>	2
	2135	Float	4	Half	2
	2135	float[2]	8	half[2]	4

Taula 1: Memòria requerida per algunes de les variables del sistema

En aquest punt sorgeix el problema que els recursos disponibles en la lògica programable no són suficients per poder implementar el disseny, concretament no hi ha suficients blocs de memòria RAM.

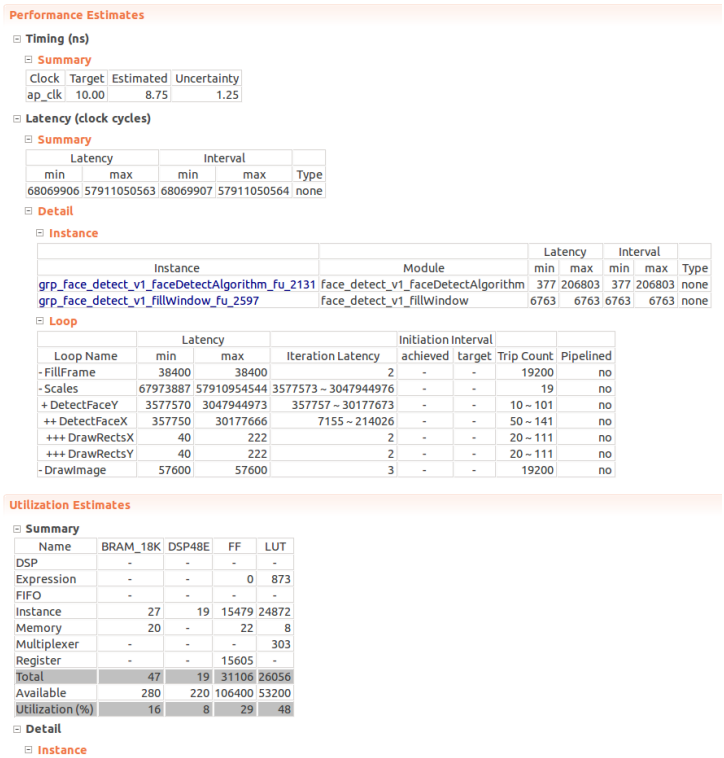
En C/C++ les variables tenen una mida fixa (8, 32 o 64 bits) i a l'hora de sintetitzar això provoca que s'utilitzin més recursos dels necessaris ja que augmenta innecessàriament els requeriments de memòria i la complexitat en les operacions. Així doncs s'ha passat d'utilitzar els tipus estàndard de C/C++ a les classes de mida arbitrària definides per Vivado HLS i amb això s'ha aconseguit reduir els recursos necessaris, tal com es pot veure en la Taula 1.

La lògica programable del xip només disposa de 280 blocs de memòria RAM de 18K bits així que la memòria necessària per implementar sBuf (el buffer on es guarden les imatges integrals per a cada escala de la imatge) continua sent massa elevada.

Amb l'objectiu de reduir la memòria necessària es va optar per, en comptes de calcular totes les imatges integrals de cop i guardar-les en un buffer, calcular la imatge integral en cada finestra i eliminar així la necessitat d'utilitzar el buffer. Amb aquesta solució es redueix la memòria necessària però s'incrementa el número de càlculs i per tant disminueix la velocitat de processament.

Un cop realitzats els canvis es va poder sintetitzar el codi. El temps que tarda Vivado HLS a realitzar la síntesi depèn de la complexitat i mida del codi, en aquest cas tarda uns quants minuts. La síntesi genera fitxers de descripció de hardware tant en VHDL com en Verilog però en comptes d'utilitzar directament aquests fitxers exportem el disseny com a IP per tal d'incloure'l directament en el disseny de blocs de Vivado.

Després de la síntesi també es generen informes, com els que es poden veure en la Figura 6, que donen informació sobre el rendiment, tant del sistema en general com el de cada bucle i funció, i els recursos utilitzats en cada operació del sistema.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.75	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
68069906	57911050563	68069907	57911050564	none

Detail

Instance

Instance	Module	Latency min	Latency max	Interval min	Interval max	Type
grp_face_detect_v1_faceDetectAlgorithm_fu_2131	face_detect_v1_faceDetectAlgorithm	377	206803	377	206803	none
grp_face_detect_v1_fillWindow_fu_2597	face_detect_v1_fillWindow	6763	6763	6763	6763	none

Loop

Loop Name	Latency min	Latency max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count	Pipelined
- FillFrame	38400	38400	2	-	-	19200	no
- Scales	67973887	57910954544	3577573 ~ 3047944976	-	-	19	no
+ DetectFaceY	3577570	3047944973	357757 ~ 30177673	-	-	10 ~ 101	no
++ DetectFaceX	357750	30177666	7155 ~ 214026	-	-	50 ~ 141	no
+++ DrawRectsX	40	222	2	-	-	20 ~ 111	no
+++ DrawRectsY	40	222	2	-	-	20 ~ 111	no
- DrawImage	57600	57600	3	-	-	19200	no

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	873
FIFO	-	-	-	-
Instance	27	19	15479	24872
Memory	20	-	22	8
Multiplexer	-	-	-	303
Register	-	-	15605	-
Total	47	19	31106	26056
Available	280	220	106400	53200
Utilization (%)	16	8	29	48

Detail

Instance

Figura 6: Captura d'un informe generat per Vivado HLS

3.2.2. Optimitzacions

La síntesis del codi no genera un hardware òptim. Per tal de sintetitzar de forma que s'afavoreixi l'execució en paral·lel del codi, reduir la latència o els recursos utilitzats es poden donar certes directives a Vivado HLS. Amb el simple fet d'aplicar directives s'aconsegueix reduir la latència del sistema però en alguns casos no serà suficient i s'haurà de reorganitzar el codi per tal de facilitar una síntesis més eficient.

En aquest punt és útil tenir en compte els resultats dels informes generats per Vivado HLS ja que ens permeten comprovar si els canvis que s'han aplicat milloren el rendiment del sistema, així com determinar quins son els bucles i funcions amb més latència i, per tant, més susceptibles a millorar.

3.2.2.1. Aplicació de directives

En un primer intent d'optimització s'ha indicat a Vivado HLS que apliqués pipeline en alguns dels bucles. Per defecte quan s'aplica pipeline en un bucle es desfan tots els bucles interiors (el mateix efecte que si s'apliqués la directiva unroll), és a dir, es crea la lògica necessària per implementar totes les iteracions dels bucles interiors en paral·lel. Per aquest motiu no es pot aplicar pipeline directament a tots els bucles ja que s'incrementaria considerablement la lògica necessària i es requeririen més recursos dels disponibles en el xip.

En aquest cas s'ha aplicat pipeline al bucle que calcula les característiques febles. Aquest bucle s'executa per a cada etapa del classificador i per a cada píxel de la imatge de cada escala, per tant una reducció en la latència d'aquest bucle repercuteix en una reducció considerable de la latència total del sistema.

Finalment també s'ha passat d'utilitzar una variable de punt flotant a punt fix, concretament de half a ap_ufixed<10,3>, per al valor de les escales. D'aquesta manera disminueix lleugerament la latència i sobretot els recursos utilitzats, ja que realitzar una operació en punt fix resulta més fàcil i menys costós en quan a recursos que una operació en punt flotant. La utilització d'una variable de punt fix ha sigut possible en aquest cas degut a que no es necessita una gran precisió per guardar el valor de les escales, cosa que no passa en la resta de variables de punt flotant.

Per defecte Vivado HLS implementa cada funció del codi com a un bloc RTL diferent i cada bloc té una interfície per comunicar-se amb l'exterior. Aquest fet fa que s'incrementi la lògica necessària ja que s'han d'implementar interfícies per a cada funció. Amb la directiva inline es pot indicar que en comptes de crear un nou bloc per una funció simplement es creï la lògica cada cop que s'utilitzi. D'aquesta manera no es crea cap interfície i es redueix la lògica necessària.

	Latència (s)	
	Mínima	Màxima
Codi inicial	0,681	579,111
Codi amb directives	0,622	110,067

Taula 2: Comparació de la latència després d'aplicar directives

Com es pot veure en la Taula 2, amb aquestes millores s'aconsegueix reduir la latència màxima (estimada per Vivado HLS) del sistema en un 80%, però tot i així el sistema continua sent molt lent i tarda uns 20 segons per processar una imatge. Bastant lluny de poder processar les imatges en temps real.

3.2.2.2. Modificació del codi

Per millorar el sistema i arribar a uns resultats que es puguin considerar acceptables s'ha optat per executar el codi en paral·lel.

S'ha mogut el bucle que itera sobre les escales dins d'una funció. L'objectiu és utilitzar múltiples vegades aquesta funció de tal manera que cada crida a la funció s'executi en paral·lel i processi diferents escales simultàniament. Així s'aconsegueix dividir la latència del sistema per un factor similar al nombre de funcions en paral·lel. En la Taula 3 es veu com la latència màxima pràcticament s'ha dividit per tres gràcies a utilitzar tres funcions en paral·lel.

Com més gran és l'escala més petita és la imatge i menys dades s'han de processar. És important que les diferents execucions de la funció estiguin equilibrades ja que la latència total del sistema dependrà de la latència de la funció més lenta. En aquest cas s'han considerat dues solucions:

- Intercalar les escales, de manera que així cada funció executa un nombre similar d'iteracions i es reparteixen els càlculs de forma més o menys equilibrada.

- Assignar un nombre diferent d'escales a cada funció, de manera que les funcions que processen les escales més petites (i han de fer més càlculs) tenen menys iteracions a realitzar.

Les dues solucions aporten resultats similars.

Vivado HLS per defecte intenta sintetitzar les funcions que es troben en el mateix nivell per a que s'executin en paral·lel. Però si les funcions comparteixen variables o si els paràmetres d'una funció depenen dels resultats d'una funció anterior aleshores les executa de forma seqüencial.

Latència (s)	
Mínima	Màxima
0,553	37,198

Taula 3: Latència del sistema utilitzant tres funcions en paral·lel

En aquest cas totes les funcions fan servir les mateixes variables, des de la imatge d'entrada fins a la de sortida passant per totes les variables del classificador. Per tant Vivado HLS no les sintetitza en paral·lel.

Per solucionar aquest problema s'han creat còpies d'aquestes variables de tal manera que cada funció disposi del seu conjunt de variables i es puguin executar en paral·lel. Aquesta solució té l'inconvenient d'incrementar el nombre de recursos, tant els blocs lògics per crear les funcions en paral·lel com la memòria RAM necessària per guardar les variables duplicades.

Degut a la falta de recursos tan sols es poden implementar tres funcions en paral·lel així que, tot i que s'aconsegueix reduir la latència, el sistema continua tardant massa temps, de l'ordre de segons.

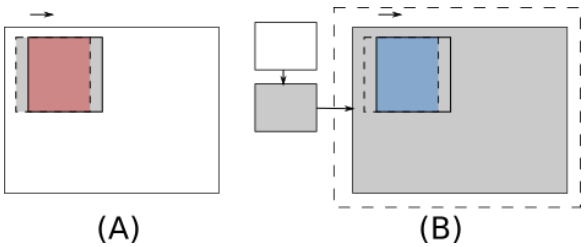


Figura 7: Representació de les diferents formes d'operar amb la imatge integral

Fins ara s'estava calculant una imatge integral de 20x20 per cada píxel de la imatge, ja que així es reduïa la memòria necessària. Aquesta solució no és gens òptima en quan a velocitat de processament ja que implica repetir una mateixa operació un gran nombre de vegades en cada píxel. En la Figura 7 (A) es pot veure aquest comportament on el requadre blanc representa la imatge original, la zona en gris la imatge integral que es calcula i la zona vermella indica els càlculs que es repeteixen d'una finestra a la següent.

En aquest punt es va provar de calcular la imatge integral tan sols una vegada per a cada escala. D'aquesta manera en comptes de calcular-la en cada finestra es calcula una sola imatge integral que es guarda en un buffer i en cada píxel tan sols s'ha d'omplir la finestra amb les dades del buffer, reduint així els càlculs necessaris considerablement. Aquest mètode està representat en la Figura 7 (B) on primer es calcula la imatge integral i després la finestra recórrer la imatge integral i no ha d'anar repetint els càlculs. En aquest cas també es solapen operacions, les de la zona blava, però son simples lectures de la imatge integral.

Al reduir els càlculs també es redueix la lògica necessària per implementar-los però s'incrementa la utilització de blocs BRAM per guardar la imatge integral.

Amb l'aproximació anterior no es podien implementar més de tres funcions en paral·lel ja que no hi havia suficients recursos, concretament LUT. Amb aquest nou plantejament tan sols es poden implementar dues funcions en paral·lel però, en aquest cas, degut a la falta de blocs BRAM. En les Taules 4 i 5 es poden veure els recursos utilitzats en cada implementació.

Nom	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expressió	-	-	0	79
FIFO	-	-	-	-
Instància	87	105	46563	44739
Memòria	54	-	0	0
Multiplexor	-	-	-	153
Registre	-	-	160	-
Total	141	105	46723	44971
Utilització (%)	50	47	43	84

Taula 4: Estimació dels recursos utilitzats en la primera implementació (sense buffer per a la imatge integral I tres funcions en paral·lel)

Nom	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expressió	-	-	0	168
FIFO	-	-	-	-
Instància	240	63	24636	17911
Memòria	14	-	0	0
Multiplexor	-	-	-	279
Registre	-	-	306	-
Total	254	63	24942	18358
Utilització (%)	90	28	23	34

Taula 5: Estimació dels recursos utilitzats en la implementació final (amb buffer per a la imatge integral i dos funcions en paral·lel)

Finalment, l'ultima optimització que s'ha aplicat consisteix en modificar l'estructura del codi per afavorir el pipeline.

L'eficiència del pipeline depèn de la latència de la operació més lenta. En aquest cas el bucle que calcula les característiques febles, al qual s'ha aplicat pipeline, realitza entre altres operacions la suma d'elements en punt flotant. Aquesta operació té una latència de 4 cicles de rellotge i limita el pipeline.

Per augmentar la velocitat del processat s'ha optat per, en comptes de realitzar la suma dins el bucle, guardar els valors en un vector i realitzar la suma fora del bucle. Per si sol el fet de moure la suma fora del bucle no aporta beneficis però ara es disposa d'un vector que conté tots els valors que s'han de sumar i, per tant, no cal sumar els elements d'un en un sinó que es poden sumar en paral·lel. En aquest cas es sumen de tres en tres i s'aconsegueix reduir la latència del sistema (Taula 6).

Latència (s)	
Mínima	Màxima
0,016	21,327

Taula 6: Latència final del sistema

3.2.3. Mesura de les característiques del sistema

Durant la fase inicial de desenvolupament del projecte s'ha prescindit de la càmera i, per poder treballar amb més comoditat i sobretot per realitzar les proves en un entorn controlat, s'han utilitzat un conjunt d'imatges fixes amb una resolució de 160x120.

Per provar el detector de cares s'ha creat un projecte amb Vivado. Concretament s'ha creat un diagrama de blocs on s'han instanciat els diferents blocs necessaris per implementar l'aplicació, tant els que s'han creat com els blocs ja inclosos en Vivado. El diagrama en qüestió és el de la Figura 8.

Les imatges es carreguen directament a la memòria DDR, el bloc "Face_detect_v4_3" és el que llegeix les imatges, realitza el processat descrit en apartats anteriors i torna a escriure el resultat en la memòria DDR.

El controlador de la memòria DDR està en la part de software del xip Zynq i, per tal de poder inicialitzar-lo i accedir a la memòria DDR s'ha d'iniciar el processador. Per aquest motiu s'ha creat una instància del bloc "ZYNQ7 Processing System" que en aquest cas també és necessari per programar en el software l'aplicació que permet calcular el temps que tarda en realitzar-se la detecció de cares. En aquest cas s'han desactivat la majoria de perifèrics del bloc "ZYNQ7 Processing System" i tan sols s'han deixat activades la comunicació USB-UART i un dels ports AXI GP per llegir el valor del comptador i AXI HP per donar accés a la memòria DDR des del hardware.

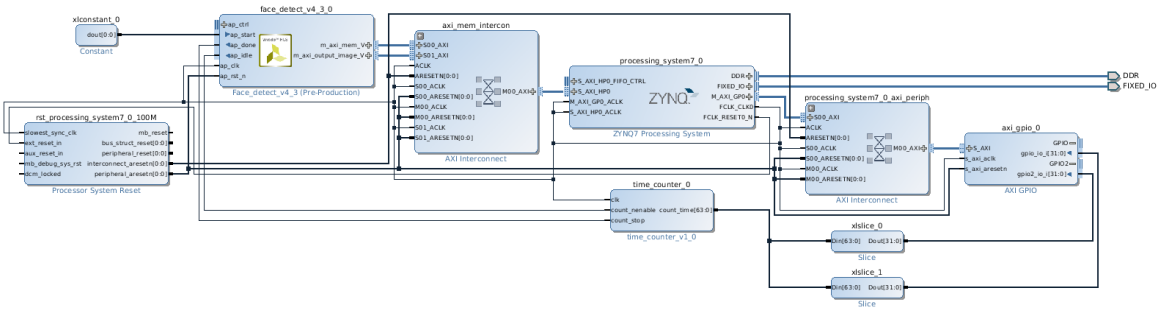


Figura 8: Diagrama de blocs del projecte utilitzat per realitzar les mesures en hardware

A més a més Vivado crea automàticament els blocs "AXI Interconnect" [4], que serveixen tant per crear una interfície entre dos ports AXI com per gestionar les transaccions de dades, i el bloc de "System Reset" que serveix per crear un senyal de reset síncron amb el senyal de rellotge.

En aquest cas s'ha indicat a Vivado HLS que utilitzés la interfície "m_axi" en els ports de la funció "Face_detect_v4_3". D'aquesta manera la nostra aplicació pot accedir directament a la memòria DDR a través dels ports AXI HP del processador.

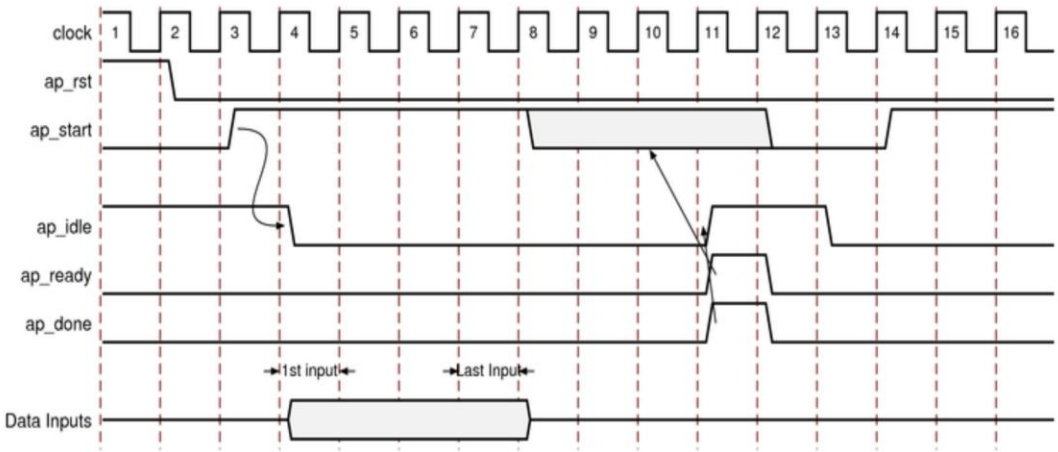


Figura 9: Esquema de la interfície ap_ctrl_hs

Per obtenir el temps real que tarda el sistema per processar una imatge s'ha utilitzat un mètode simple basat en un comptador i la interfície de control, “ap_ctrl_hs”, del bloc “Face_detect_V4_3” creada automàticament per Vivado HLS i que se’n pot veure el funcionament en la Figura 9.

El comptador té una longitud de 64 bits i la seva sortida és un registre. El funcionament del comptador és simple: el comptador s’incrementa en cada cicle de rellotge mentre el senyal d’activació és baix. A més consta d’un senyal d’aturada que quan s’activa guarda el valor actual del comptador en el registre de sortida i reinicia el comptador a zero.

La interfície “ap_ctr_hs” implementa els senyals “ap_idle” que es manté baixa mentre el bloc al qual pertany està realitzant la seva funció i el senyal “ap_done” que s’activa quan el bloc ha acabat la seva funció. Per tant, el senyal “ap_idle” es fa servir com el senyal d’activació del comptador i el senyal “ap_done” com al senyal d’aturada.

En aquest cas el comptador està implementat en VHDL i s'ha creat una IP, “time_counter”, que s’ha inclòs en el disseny de blocs de Vivado.

Tornant al disseny de blocs, la sortida del comptador està connectada al bloc “AXI GPIO” [6], que crea una interfície entre la lògica i el processador, de tal manera que es pot accedir al valor del comptador a través d'una aplicació en software. El processador té designades un rang d’adreces per a la comunicació amb el software mitjançant els ports AXI GP [8] i des de Vivado es pot veure quin rang d’adreces s’ha assignat per a la configuració i utilització del bloc “AXI GPIO”.

S’ha creat una aplicació en software que simplement inicialitza el bloc “AXI GPIO” i dins d'un bucle infinit va llegint constantment el valor del comptador amb un temps d'espera d'un segon entre lectures. Donat que es treballa amb una imatge fixa aquest mètode és suficient ja que, encara que el valor del registre del comptador s’actualitzi abans que es produeixi la lectura, com que la imatge serà la mateixa el valor del comptador també serà el mateix.

Finalment l’aplicació envia els resultats a l’ordinador mitjançant la interfície USB-UART de manera que així es poden visualitzar els resultats per consola.

3.2.4. Projecte final

En el projecte final es prescindeix del processador i tot el sistema està implementat en hardware. Per tant el disseny del sistema, corresponent a la Figura 10, canvia lleugerament respecte la versió de prova.

El bloc “ov7670_top” configura la càmera i guarda les imatges capturades en una memòria BRAM de doble port. El bloc “Face_detect_full_hardware” llegeix les dades d'aquesta memòria, implementa l'algoritme i guarda les dades en una nova memòria BRAM, també de doble port. Finalment el bloc “BRAM_to_VGA” llegeix les dades de la segona memòria BRAM i les mostra per pantalla.

Aquest nou disseny treballa amb tres freqüències diferents. El bloc “ov7670_top” funciona a 50 MHz, el bloc “BRAM_to_VGA” a 25.175 MHz i el bloc “Face_detect_full_hardware” a 100 MHz, tot i que la freqüència màxima que suporta el xip és de 667 MHz.

A més dels blocs principals que s'han descrit el disseny inclou el bloc “Clocking Wizard”, que genera els rellotges de les diferents freqüències a partir del rellotge extern de la placa (de 100 MHz) i el bloc “Video Timing Generator” que genera els senyals de vídeo

per al bloc “BRAM_to_VGA”. També hi ha un bloc “Constant”, equivalent a un '1' lògic, que simplement activa el bloc “Video Timing Controller” i en desactiva el reset.

Un píxel capturat per la càmera té una mida de 12 bits, 4 bits per a cada component en format RGB. Degut a la poca memòria disponible tan sols es guarda un dels components i, a causa d'això, la imatge final es mostrarà en blanc i negre. El bloc “Slice” s'encarrega d'extreure aquest 4 bits i ajustar la mida per poder connectar correctament els diferents blocs.

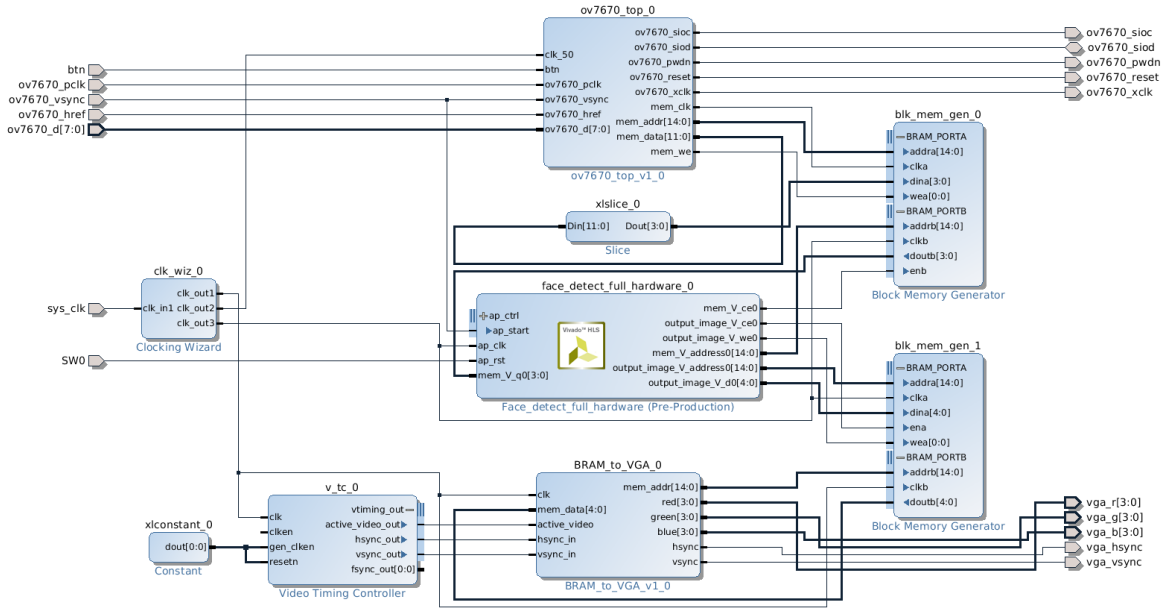


Figura 10: Diagrama de blocs de la versió final

Per poder implementar aquesta solució s'ha modificat la interfície del bloc “Face_detect_full_hardware”. Ara, en comptes d'utilitzar una interfície per al port AXI HP i poder accedir a la memòria DDR, utilitza una interfície ap_memory amb la qual es pot connectar directament a un bloc BRAM.

Com s’ha comentat, també s’han afegit dos blocs de memòria BRAM de doble port. Un d'ells guarda la imatge d'entrada capturada per la càmera i l'altre guarda la imatge de sortida que es mostrarà per pantalla.

Un cop sintetitzat i implementat el projecte s’obté el fitxer bitstream amb el qual es pot programar el xip. Finalment, en la Figura 11 es poden veure els recursos utilitzats per aquesta implementació. En aquest cas correspon a la utilització real obtinguda després de sintetitzar i implementar el disseny complert.

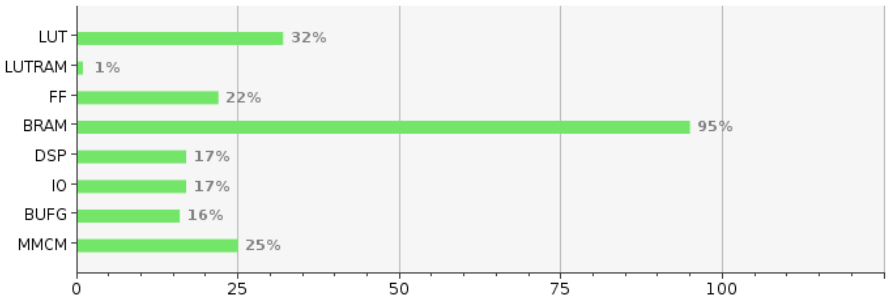


Figura 11: Utilització dels recursos en la versió final

3.3. Implementació en software

3.3.1. Adaptació del codi

En aquesta aplicació la implementació de l'algoritme no s'havia de fer sobre un sistema operatiu sinó directament sobre el processador ARM, per aquest motiu no es podia utilitzar directament el codi OpenCV. En un primer moment es va estudiar la possibilitat de compilar la llibreria OpenCV per a ARM però es va descartar debut a la complexitat que comportava i es va optar per utilitzar el codi simplificat que s'ha descrit en apartats anteriors.

Per tal d'implementar l'algoritme en el processador ARM es va haver de prescindir del sistema de fitxers i per poder carregar les valors del classificador es van utilitzar els mateixos fitxers de capçalera creats per a la implementació en hardware.

En aquest cas en l'SDK s'ha de configurar l'script de l'enllaçador[4] ja que per defecte aquest assigna tan sols 1 KB de memòria tant per la pila com pel *heap* i aquesta memòria és insuficient per guardar les variables del codi. En aquest cas s'ha assignat una memòria de 20 MB tant per la pila com pel *heap*. En la Figura 12 es mostra la finestra de configuració de l'script.

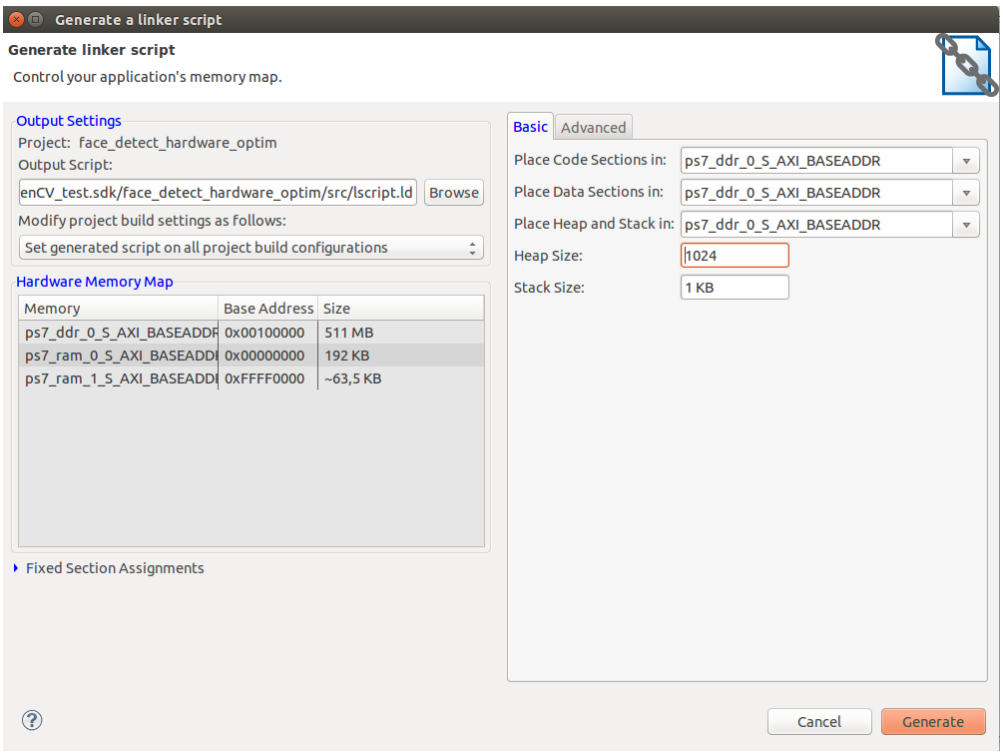


Figura 12: Captura de la finestra de configuració de l'script de l'enllaçador en l'SDK

3.3.2. Mesura de les característiques del sistema

Similarment com s'ha fet per a la versió en hardware s'ha creat un projecte amb Vivado en el qual s'ha instanciat el bloc del processador "ZYNQ7 Processing System". Aquest bloc també permet ajustar la freqüència del processador que en aquest cas s'ha deixat a

[4] Linker script

667 MHz, que és el valor per defecte i el màxim del xip. En aquest cas, per realitzar proves, no es necessita cap altre bloc tot i així s'han creat uns blocs addicionals per poder visualitzar per pantalla els resultats com es pot veure en la Figura 13 (aquests nous blocs també es poden utilitzar en la mesura de la implementació en hardware descrita anteriorment).

Així, es va crear el bloc "Ddr_extract_frame_line" amb Vivado HLS per llegir les dades de la memòria DDR. El bloc "Ddr_extract_frame_line" llegeix una línia de la imatge i la guarda en un bloc BRAM. En aquest cas el bloc "frame_to_vga" llegeix les dades del bloc BRAM i les mostra per pantalla i, a més a més, indica al bloc "Ddr_extract_frame_line" quina línia de la imatge ha de llegir. També s'ha incorporat un bloc "AXI GPIO" que serveix per indicar al bloc "Ddr_extract_frame_line" la posició de memòria on està guardada la imatge.

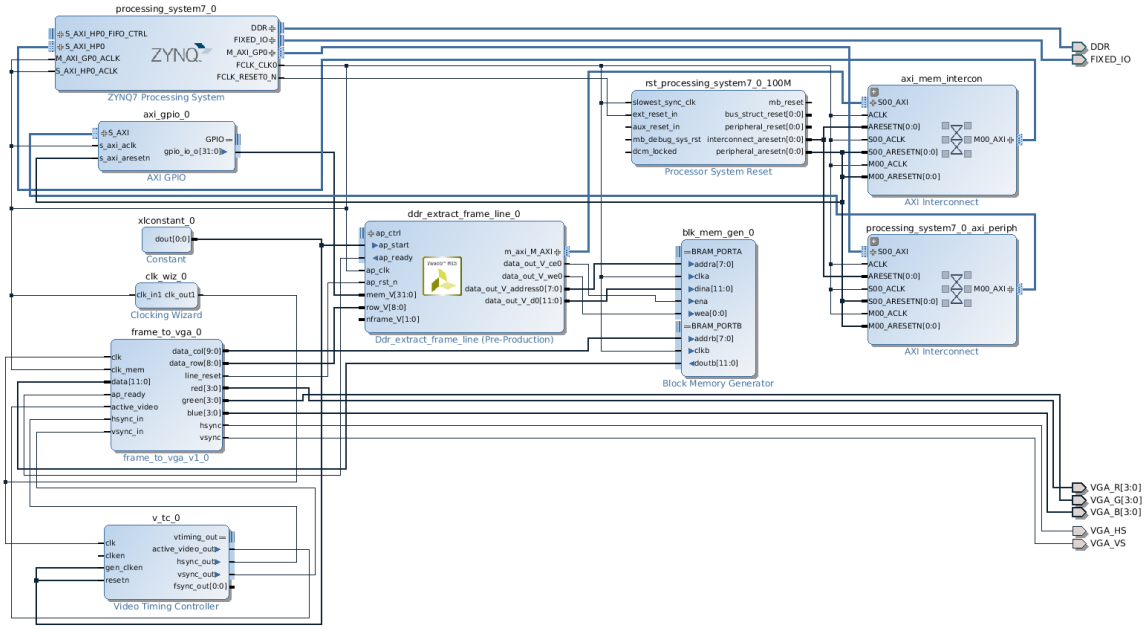


Figura 13: Diagrama de blocs utilitzat en la versió software

Un cop implementat el projecte en Vivado s'ha de passar a l'SDK. En aquest cas per mesurar el temps que tarda l'aplicació s'ha utilitzat directament el Timer del dispositiu [1]: es mesura la diferència entre el valor del comptador abans i després d'aplicar l'algoritme i a partir d'aquí s'obté el temps en segons.

Si es volen visualitzar els resultats també s'han d'inicialitzar els registres de l'AXI GPIO i escriure-hi el valor de la posició de memòria on es troba la variable que guarda la imatge de sortida.

3.4. Interfície de càmera i pantalla

Tant la interfície de la càmera com la pantalla estan implementades en el hardware de la placa.

Per implementar la configuració i la captura dels píxels de la càmera s'ha utilitzat el codi creat per Mike Field [11] que proporciona diferents elements que permeten configurar els registres de la càmera així com realitzar la captura de la imatge. En aquest cas

simplement s'ha creat el fitxer "ov7670_top.vhd" per agrupar els diferents elements necessaris en un sol bloc.

La interfície VGA s'ha implementat en VHDL. En aquest cas es parteix dels senyals de sincronització de vídeo, generats mitjançant el bloc "Video Timing Controller" de la mateixa llibreria Vivado. A partir dels senyals de vídeo el bloc VGA genera les adreces mitjançant les quals obté els píxels que mostrarà per pantalla.

Els senyals de vídeo que es fan servir són per a una resolució de 640x480 però la imatge utilitzada és de 160x120. En aquest cas es mostra la imatge en la zona superior esquerra de la pantalla i la resta es manté en negre.

3.5. XSDB

El Xilinx *System Debugger* (XSDB) [9] és una eina inclosa en l'SDK que permet analitzar què està passant mentre un programa s'executa. Es poden establir punts d'aturada per aturar el processador, realitzar una execució pas a pas del programa, veure el valor de les variables o els continguts de la memòria del sistema.

En aquest cas s'ha utilitzat XSDB principalment per accedir a la memòria DDR i escriure-hi les imatges de test per avaluar el funcionament del detector. Per fer-ho s'ha accedit al XSDB mitjançant la consola del sistema i s'han realitzat les operacions següents:

1. Establir una connexió amb el xip:

   xsdb% connect

   Donat que la connexió amb la placa es realitza de forma local no s'utilitza cap opció de configuració i es deixa el comportament per defecte.

2. Seleccionar el dispositiu a programar:

   xsdb% targets 1

   Determina el dispositiu actiu, és a dir, quin processador es vol utilitzar.

3. Escriure en la memòria:

   xsdb% mwr –bin –address-space AP0 –file data.bin 0x00200000 19200

   En aquest cas -bin indica que s'escriuran les dades des del fitxer binari especificat per –file. La opció –address-space AP0 és necessària per accedir a la memòria DDR. Finalment en aquest cas 0x0020000 és l'adreça on s'escriurà i 19200 el nombre de dades a escriure.

4. Llegir de la memòria:
   xsdb% mrd –bin –address-space AP0 –file data.bin 0x00200000 19200
   El comportament és el mateix que per excriure però en aquest cas –file indica el fitxer on es guardaran les dades.

A més, per visualitzar fàcilment les resultats s'ha creat un script amb Python per generar la imatge a partir del fitxer binari.

4. Resultats

Els resultats que s'han obtingut utilitzant les imatges de prova es poden veure en la Figura 14, on la primera fila (A) correspon a l'algoritme implementat en software i la segona fila (B) a la implementació en hardware. Com es pot observar, les deteccions de les dues implementacions són pràcticament iguals. Finalment en la Figura 15 es veuen els resultats de l'aplicació en hardware executant-se en temps real.

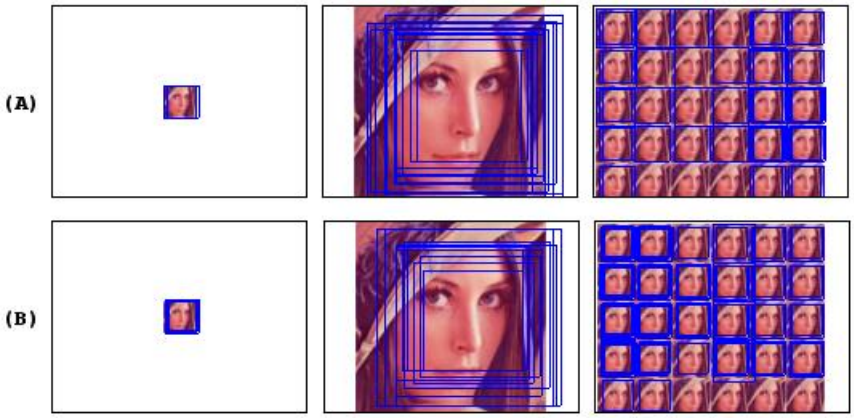


Figura 14: Resultats obtinguts en les diferents implementacions

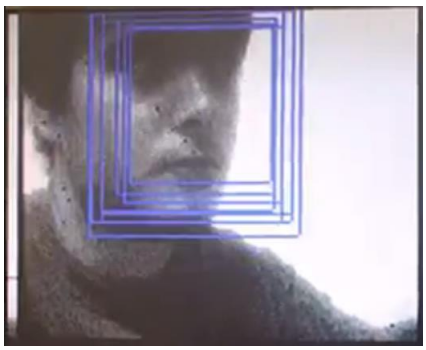


Figura 15: Captura de la implementació en hardware executant-se en temps real

A continuació s'analitzaran amb més profunditat els resultats obtinguts comparant les diferents implementacions de l'algoritme de detecció de cares en quan a velocitat de processament i potència consumida.

4.1. Comparativa del rendiment

Com es pot observar en la Taula 7, el processat de les dades en la implementació en software és molt lent i, per tant, no és apte per a aplicacions que requereixin una detecció en temps real. D'altra banda amb l'aplicació en hardware s'aconsegueix accelerar considerablement el processat, entre 33 i 93 vegades en les diferents imatges utilitzades, fet que permet processar les imatges a entre 5 i 2,8 FPS, suficient per a una detecció en temps real.

	Temps (s)		
	Una cara petita	Una cara gran	Moltes cares petites
HW	0,20	0,30	0,35
SW	6.60	27,87	32,71

Taula 7: Temps de processat de les diferents implementacions

També s'ha mesurat el consum de la placa en les diferents implementacions. Els resultats, disponibles en la Taula 8, mostren que la implementació en hardware consumeix significativament menys que la versió en software. Concretament el consum en hardware, descomptant la potència en repòs, és un 60% inferior respecte la versió en software.

	Potència mesurada (W)
Placa en repòs	2,22
Implementació HW	2,81
Implementació SW	3,72

Taula 8: Comsum de potència de les diferents implementacions

A partir dels resultats obtinguts es pot determinar que una implementació en hardware permet accelerar significativament el processament de les dades en sistemes de càlcul complexos.

A més a més s'ha observat que amb dispositius de gamma mitja es pot arribar a realitzar un processat complex, com el de detecció de cares, en temps real gràcies a la utilització d'un hardware dedicat. En el cas de sistemes autònoms, una implementació en hardware no tan sols és beneficiós en quant a velocitat de processament sinó també en quant a la reducció del consum, augmentant així l'autonomia del sistema.

4.2. Anàlisis del desenvolupament

Durant el desenvolupament del projecte s'ha pogut comprovar que la utilització de Vivado HLS permet accelerar considerablement el procés de disseny i implementació d'un sistema en hardware. Tot i així el desenvolupament en software continua sent molt més ràpid i flexible.

Finalment també s'ha utilitzat el codi en C/C++ modificat per Vivado HLS el qual, un cop sintetitzat, permet executar l'aplicació en hardware en temps real i s'ha executat en el processador per comprovar com afecten els canvis aplicats al rendiment del sistema.

	Temps (s)		
	Una cara petita	Una cara gran	Moltes cares petites
Software original	6,60	27,87	32,71
Software modificat	46,53	61,38	71,47

Taula 9: Temps de processat del software original i modificat

A partir dels resultats representats en la Taula 9 es veu que les modificacions aplicades empitjoren considerablement el rendiment del sistema.

Es pot determinar doncs que per realitzar aplicacions amb Vivado HLS no s'ha de plantejar el sistema com una aplicació en software sinó que s'ha de plantejar des del punt de vista del hardware.

5. Costos

L'objectiu d'aquest projecte no era la realització d'un prototipus sinó la comparació entre dues implementacions d'un mateix sistema. Per aquest motiu es realitzarà una avaluació per separat dels costos associats a cada implementació de manera que es puguin comparar.

Eines de desenvolupament	Unitats	Cost/Unitat	Cost
ZedBoard	1	420,00 €	420,00 €
Càmera	1	6,90 €	6,90 €
Monitor	1	60,00 €	60,00 €
Ordinador	1	799,00 €	799,00 €
		Subtotal	1.285,90 €
Software	**Unitats**	**Cost/Unitat**	**Cost**
Vivado Design Suite	1	Gratuït	- €
		Subtotal	0,00 €

Taula 10: Costos associats a les eines de desenvolupament

Desenvolupament Hardware	Hores	Cost/Hora	Cost
Investigació	120	10,00 €	1.200,00 €
Desenvolupament	320	10,00 €	3.200,00 €
Optimització	200	10,00 €	2.000,00 €
		Subtotal	6.400,00 €

Taula 11: Costos associats al desenvolupament en hardware

Desenvolupament Software	Hores	Cost/Hora	Cost
Investigació	80	10,00 €	800,00 €
Disseny i implementació	100	10,00 €	1.000,00 €
		Subtotal	1.800,00 €

Taula 12: Costos associats al desenvolupament en software

Els costos totals per al desenvolupament del projecte són de 9.485,90 €. S'observa a més que els costos del desenvolupament en hardware són quatre vegades més que els costos del desenvolupament en software. Aquesta diferència està en el cost del personal causada per un increment considerable en el temps de desenvolupament de la versió hardware respecte la versió software.

6. Conclusions i futur desenvolupament:

En aquest projecte s'ha agafat com a punt de partida una aplicació que requereix un processat intens de les dades i s'ha aconseguit accelerar considerablement mitjançant una implementació en un hardware dedicat. Aconseguint en aquest cas una detecció de cares en temps real i complint els objectius inicials proposats.

A més s'ha fet una introducció a Vivado HLS que és una eina molt útil per accelerar el disseny en hardware. Sense Vivado HLS aquest projecte possiblement no s'hagués pogut dur a terme en el temps establert. Tot i així Vivado HLS és una eina molt completa i durant el desenvolupament d'aquest projecte no hi ha hagut temps per estudiar-la a fons i aprofitar-la al màxim.

Una de les conclusions a les que s'ha arribat és que si es vol un sistema de baix consum, la millor solució és una implementació en hardware. Per altra banda si el que es busca és una millor velocitat de processat la millor solució pot no ser una implementació en hardware sinó una implementació hibrida que combini hardware i software. Com s'ha vist durant el desenvolupament del projecte el principal problema de la implementació en hardware és la falta de recursos, concretament en la implementació final si no fos per la falta de memòria es podrien duplicar (fins i tot triplicar) el nombre de funcions que s'executen en paral·lel i reduir a la meitat el temps de processat. Per tant sembla que utilitzar un software pot ser una bona idea per ajudar a gestionar millor la memòria i maximitzar així els recursos que el hardware dedica al processat de les dades.

6.1. Futur desenvolupament

En un futur desenvolupament es podria augmentar la freqüència de rellotge i observar l'impacte que causa tant en velocitat de processament com en potència consumida. Addicionalment es podrien utilitzar els recursos disponibles en la ZedBoard per implementar un sistema híbrid que combini la lògica programable amb el processador i, si bé en qüestió de consum no aportarà beneficis, és possible que es produeixi un increment respecte els resultats obtinguts en aquest projecte en quant a velocitat de processament.

Abans de portar a terme la implementació d'un sistema híbrid s'hauria de dedicar cert temps a l'estudi de la interconnexió del processador i la lògica programable, la utilització de les diferents interfícies AXI i la gestió de les transaccions de dades, ja que són temes que poden arribar a ser bastant complexos i provocar errors si no s'implementen de forma correcta.

Bibliografia

[1] ARM Ltd, “Cortex-A7 MPCore: Technical Reference Manual”, p. 70, 2013.

[2] Xilinx Inc., “Zynq-7000 All Programmable SoC Overview”, *DS190,* Gener 2016.

[3] Xilinx Inc., All Programmable 7 Series Product Tables and Product Selection Guide”, 2015

[4] Xilinx Inc., “AXI Interconnect v2.1”, *PG059*, 2015.

[5] Itseez, “OpenCV.” [Online]. Disponible a: http://opencv.org/. [Últim accés: 30-Maig-2016].

[6] Xilinx Inc., “AXI GPIO v2.0”, *PG144,* 2015.

[7] P. Viola and M. Jones, “Rapid object detection using a boosted cascade of simple features,” *Computer Vision and Pattern Recognition*, 2001.

[8] Xilinx Inc., “Zynq-7000 All Programmable SoC Technical Reference Manual”, *UG585,* p. 40,113, 2014.

[9] Xilinx Inc., “Embedded System Tools Reference Guide”, *UG*1043, 2009.

[10] Xilinx Inc., “Vivado Design Suite User Guide”, *UG903*, 2015.

[11] M. Field, “Zedboard OV7670,” 2013. [Online]. Disponible a: http://hamsterworks.co.nz/mediawiki/index.php/Zedboard_OV7670. [Últim accés: 25-Maig-2016].

[12] ARM Ltd., “Cortex-A series.” [Online]. Disponible a: https://www.arm.com/products/processors/cortex-a/index.php. [Últim accés: 20-Maig-2016].

[13] ARM Ltd, “Cortex-A9 Processor.” [Online]. Disponible a: http://www.arm.com/products/processors/cortex-a/cortex-a9.php. [Últim accés: 20-Maig-2016].

[14] Xilinx Inc., “Vivado Desing Suite HLx Editions.” [Online]. Disponible a: http://www.xilinx.com/products/design-tools/vivado.html. [Últim accés: 21-Abril-2016].

Annex

Inclòs amb aquest treball s'entrega el fitxer "TFG_Comparativa_Deteccio_Cares.zip". Aquest fitxer comprimit conté els diferents codis, IPs i imatges utilitzades seguint la següent estructura:

- /Imatges_Test
  - data.bin
  - data1.bin
  - data2.bin
  - Lenna_160_120.png
  - Lenna_160_120_1.png
  - Lenna_160_120_2.png
- /Implementacio_Hardware
  - /Bitstream
    - face_detect.bit
  - /Codi_HLS
    - Scales.hpp
    - SimpleFeature_v2.hpp
    - SimpleNode.hpp
    - SimpleStage.hpp
    - top.cpp
    - top.hpp
  - /Ips
    - user.org_user_frame_to_vga_1.0.zip
    - user.org_user_ov7670_top_1.0.zip
    - user.org_user_time_counter_1.0.zip
    - xilinx_com_hls_ddr_extract_frame_line_1_0.zip
    - xilinx_com_hls_face_detect_full_hardware_1_0.zip
- /Implementacio_Software
  - HaarDetector.cpp
  - HaarDetector.hpp
  - main.cc
  - SimpleFeature.hpp
  - SimpleNode.hpp
  - SimpleStage.hpp

La carpeta “Imatges_Test” conté les imatges de prova que s’ha utilitzat, tant en format png com en binari.

La carpeta “Implementacio_Hardware” conté:

- El fitxer “face_detect.bit”, necessari per programar el disseny en hardware al xip.
- El codi utilitzat en Vivado HLS per a la versió final del projecte. En aquest cas “top.hpp” i “top.cpp” defineixen la funció principal i els altres fitxers són els que contenen les dades del classificador.
- Les diferents IPs que s’han creat. Vivado genera les IPs com un fitxer zip i dins d’aquest fitxer hi ha els diferents codis en VHDL i altres fitxers necessaris per a la seva utilització.

Finalment la carpeta “Implementacio_Software” conté el codi necessari per a la implementació en software de l’algoritme. “HaarDetector.cpp” i “HaarDetector.hpp” contenen les funcions que implementen el classificador, “main.cc” conté la funció principal del programa i els altres fitxers contenen les dades del classificador.

Glossari

ARM – *Advanced RISC Machines*

BRAM – *Block RAM*

DDR – *Double Data Rate*

DSP – *Digital Signal Processor*

FF – *Flip-Flop*

FPGA – *Field Programmable Gate Array*

FPS – *Frames per Second*

HLS – *High-Level Synthesis*

IDE – *Integrated Development Environment*

LTS – *Long Time Support*

LUT – *Look Up Table*

PL – *Programmable Logic*

PS – *Processing System*

RAM – *Random Access Memory*

RTL – *Register Transfer Level*

SDK – *Software Development Kit*

SoC – *System on Chip*

UART – *Universal Asynchronous Receiver/Transmitter*

USB – *Universal Serial Bus*

VGA – *Video Graphics Array*